First order logic
What is new compared to propositional logic?

• We have a collection of things.
• We call this the domain of discourse.
• We have “predicates” that state properties about the items in the collection.
• We can quantify statements in the logic
  – Universal quantification – for all x ...
  – Existential quantification - there exists x ....
Examples

• All natural numbers are either even or odd
  – What is the domain of discourse?

• In the Family tree example (from the FiniteSet code), no one is a descendant of themselves.
  – What is the predicate?

• Addition is commutative
  – What is the domain?
  – What is the predicate?
Observation

• Many logics have these distinctions
  – A domain of discourse
  – A set of predicates over the domain
    • Some logics add functions over the domain as well as predicates
  – A set of connectives (and, or, not, etc)
  – A set of quantifiers (forall, exists)
    • Some logics (e.g. temporal) add more quantifiers

• How does propositional logic fit in this framework?
First order logic

• A domain of discourse
• Terms over the domain
  – A minimum of variables
  – Sometimes constants
  – Some times functions
• Formulas
  – Predicates P(term, ..., term)
  – Connectives (and, or, not, implies)
  – Quantifiers (for all, exists)
Formulas and Terms

- A First-order logic is a parameterized family of logics
  - Parameters
    - Constants (c)
    - Function symbols (f)
    - Predicate symbols (p)
- $L(c,f,p)$ is a logic for concrete c, f, and p
- Quantifiers are bound in formula, but name individuals used in terms
- Predicates are atomic elements of formulas but are applied to terms
- Both functions and predicates are applied to a fixed number of arguments, called their arity.
- Constants are functions of arity 0 (implies $C \subseteq F$)
Definition of Terms for L(C,F,P)

• Let C be a subset of F
• Any variable is a term
• If c is a nullary function then c is a term
• If $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term
• Nothing else is a term
Atomic formula of $L(C,F,P)$

- If $p$ is an $n$-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an atomic formula.
- True and False are atomic formula.
Inductive Formula over L(C,F,P)

• If \( w \) is an atomic formula, then \( w \) is a Formula
• If \( w \) is a formula, then \( \neg w \) is a formula
• If \( w \) and \( v \) are formula then so are
  – \( w \land v \)
  – \( w \lor v \)
  – \( w \rightarrow v \)
• If \( x \) is a variable and \( w \) is a formula then so are
  – Forall \( x . w \)
  – Exists \( x . w \)
Free and bound variables

• Quantifiers add complexity because they bind variables in a certain scope.
• Some variables are free because they are not in scope of any quantifier.
• A closed formula (sometimes called a sentence) has no free variables.
• A formula with at least one free variable is called open.
Truth of Formula

• We will eventually get around to defining the truth or falsehood of a formula.
• These concepts usually apply to only “closed formula”
• For an open formula we must be more precise by what we mean by the free variables.
We will illustrate with a Haskell Program

• Consists of many files
  – Term.hs
  – Formula.hs
  – Subst.hs
  – Print.hs
  – etc
data Term f v = Var v
  | Fun Bool f [Term f v] deriving Eq

variables :: Term f v -> [Term f v]
variables (Var v) = [Var v]
variables (Fun s n ts) = concat (map variables ts)

newVar :: Int -> Term f String
newVar n = Var ("?" ++ intToString n)

newFun :: Int -> [Term String v] -> Term String v
newFun n ts = Fun True ("_" ++ intToString n) ts
Substitution

• Substitution replaces a variable with a term
• Its is a natural operation, but is subtle because the quantifiers bind variables.
• Variables in the scope of a quantifier should not be substituted
• Substitution is a monadic function
  \[-\text{type } \text{Subst } v \text{ m } = v \rightarrow m \text{ v}\]
• Read \( t \gg= s \) as the image of \( t \) under substitution \( s \)
Subst

type Subst v m = v -> m v

emptySubst :: Monad m => Subst v m
emptySubst v = return v

-- Substituting the variable v with the term t

(\-\-\-\) :: (Eq v, Monad m) => v -> m v -> Subst v m
(v \-\-\-\- t) v' | v == v' = t
                | otherwise = emptySubst v'

-- Composing two substitutions

(\=\=\) :: Monad m => Subst v m -> Subst v m -> Subst v m
s1 \=\= s2 = (s1 =<<) . s2

-- Removing a variable from a substitution

(\/-\-) :: (Eq v, Monad m) => v -> Subst v m -> Subst v m
(v \/-\- s) v' | v == v' = return v'
               | otherwise = s v'
data Formula r f v = Rel r [Term f v]
    | Conn Cs [Formula r f v]
    | Quant Qs v (Formula r f v)

deriving Eq

data Qs = All | Exist deriving Eq

data Cs = And | Or | Imp
    | T | F | Not

deriving Eq

subst :: Eq v => (v -> Term f v) -> Formula r f v -> Formula r f v

subst s (Rel r ts)  = Rel r (map (s =<<) ts)
subst s (Conn c fs) = Conn c (map (subst s) fs)
subst s (Quant q v f) = Quant q v (subst (v |\:-\:/>| s) f)